LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-610032

Tuning the LULESH Mini-app for Current and Future Hardware

I. Karlin, J. McGraw, J. Keasler, B. Still

January 14, 2013

NECDC 2012
Livermore, CA, United States
October 22, 2012 through October 26, 2012

Disclaimer

Tuning the LULESH Mini-app for Current and Future Hardware(U)

Ian Karlin[1], **Jim McGraw**[1], **Jeff Keasler**[1], **Bert Still**[1]
[1]*Lawrence Livermore National Laboratory, Livermore, CA*

Abstract

Current and emerging architectures present challenges for scientific programmers. Memory capacity and memory bandwidth are increasing slower than floating point capability. Also, the clock speed of individual processors is no longer increasing while available on-node parallelism explodes in the form of increased cores per processor, threads per core and vector unit width. To effectively use the new hardware, programmers must exploit greater parallelism while using relatively less memory. In light of this need, we tuned the proxy application, Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH), for better on-node performance achieving 62% fewer memory reads, a 19% smaller memory footprint and an eight-fold increase in fully vectorized floating point operations. Tests on a Linux cluster containing Sandybridge processors and a BlueGene Q machine showed serial runtime decreased by up to 54% and parallel runtime reductions of up to 65% versus the baseline code versions. Strong scalability increased from 21.69x to 32.23x on a 16 core 64 thread BG/Q node, and from under 5x to over 6x on a single socket of the Linux node. We also reduced the serial section runtime of the parallel proxy to less than 0.1% of total execution time. These optimizations are being put into a subset of ALE3D, one the application LULESH most closely models, resulting in overall runtime reductions of up to 20% for the same test sedov problem. (U)

Introduction

Hydrodynamics is widely used to model continuum material properties and material interactions in the presence of applied forces [10]. The hydrodynamics portion of an explicit timestepping multi-physics application might typically consume one third of the total runtime. To provide a simple, but still full-featured, hydrodynamics problem to test various tuning techniques and different programming models, the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) mini-app was created as one of five challenge problems in the DARPA UHPC program [1]. Hydrodynamics was chosen as one of the challenge problems because it had been measured to consume 27% of all DoD data center computing resources. LULESH solves the sedov problem, modeling one octant of a symmetrical blast wave. The reason to use mini-apps like LULESH is they can be used to explore various programming techniques that would take substantially longer to test a larger code [11, 14]. The successful lessons learned from the mini-app exploration can then be applied to a full application.

We are using LULESH to test optimization techniques and programming practices that are likely to increase code performance on current and future architectures. In particular our focus is on single node performance, which is where the hardware is most rapidly evolving due to exponentially expanding parallelism, decreasing relative memory bandwidth and less memory per thread [12]. For example, in terms of parallelism, the two newest architectures at Lawrence Livermore National Laboratory have sixteen cores per node, four hardware threads per core, and each core can execute eight floating point instructions per cycle for 128 way per cycle floating point parallelism. Memory bandwidth on each machine delivers less than one half of a byte of data in the amount of time it takes to perform one floating point operation. Also, the BG/Q based Sequoia machine contains only 256 MB of main memory per hardware thread, which is less than the size of some multi-physics application executables.

To achieve these goals we applied five optimizations to LULESH: loop fusion, array contraction, increased vectorization, NUMA aware allocation and global allocation of temporary work arrays. These changes reduced last level cache misses by over 62%, the maximum global state size of the program by 19%, serial section runtime to less than 0.1% of the overall runtime, and increased the percentage of floating point operations that vectorize from 10% to 87%. Tests on a Linux cluster containing Sandybridge processors and a BlueGene Q machines showed serial runtime decreased by up to 54% and up to a 65% parallel runtime reduction versus the baseline code versions. Strong scalability increased from 25x to 36x on a 16 core 64 thread BG/Q node, and from under 5x to over 6x on a single socket of the Linux node. From the lessons learned tuning LULESH, we are porting changes back to a more complex proxy-app xALE, which contains many of the features of the full ALE3D hydrodynamics application [15]. As of this writing, we have improved xALE performance by up to 25% for the same problem.

Overall, our paper makes the following novel contributions:

- We show by combining performance tuning techniques, the runtime of a hydrodynamics proxy-app can be reduced by a factor of two or more. We demonstrate that these techniques increase the strong scalability of the code by reducing the serial component of runtime and data motion, taking better advantage of on node parallelism, and reducing the application's memory footprint by 19%.
- We apply the tuning techniques that demonstrated success on LULESH back to a more full featured application and achieve a 20% runtime reduction. By applying these tuning techniques to kernels not represented in LULESH, we show that lessons learned in a proxy-app exploration have widespread applicability to modeled codes.
- We explore the relative difficulty of applying tuning techniques, that were successfully applied to the proxy-app, to a real application. Throughout the paper we discuss the advantages and disadvantages of applying these techniques to large multi-physics applications. Including, but not limited to their performance impact, code maintainability and auto-tunability.

The rest of this paper is organized as follows: Section 2 describes the tuning techniques we applied to LULESH and how they vary between BlueGene and Intel based systems. It also discusses the applicability of these techniques to multi-physics codes. Then Section 3 analyzes the performance gains from these techniques. It includes a comparison of the impact of the optimizations on different loops and a cross

comparison of relative performance impacts on different architectures. Section 4 describes how we apply the tuning techniques used in the proxy application to improve the performance of a real application. We analysis where tuning xALE is of similar difficulty or more complex than tuning LULESH. Then Section 5 describes the performance impacts of tuning xALE. Finally, we present conclusions and describe and describe future directions for our work in Section 6.

Tuning LULESH

To tune LULESH for modern CPUs we performed multiple optimizations: loop fusion, global allocation, increased vectorization and non-uniform memory (NUMA) aware allocation. In this section, we describe the optimizations we performed on LULESH. We discuss how and why they improve performance and how the optimizations interact with one another. Our discussion also includes how they can be applied, or the additional support needed, to apply them to a large multi-physics code and achieve portable performance.

Loop Fusion

Loop fusion is an optimization that involves combining multiple loops over the same iteration space into a single loop. When loops are fused, arrays that are only used to store data in the first loop that is consumed in the second loop can be contracted to scalar temporaries [6]. Combining loops that access the same data structures allows fewer reads and writes to memory structures, however, it can stress other resources [7]. By increasing the amount of data accessed in the same loop iteration, loop fusion can increase register pressure and the amount of data needed to stay within cache for good performance. Also, the number of data streams being read can increase and when the number becomes larger than the number of hardware stream pre-fetchers performance can decline. Finally, most compilers only vectorize loops if the whole loop is vectorizable so fusing a loop that is non-vectorizable with a vectorizable loop results in a non-vectorizable loop.

In LULESH we were able to fuse the 45 loops that occur each timestep to twelve loops, which also reduced the number of OpenMP parallel regions from 30 to twelve. We reorganized where checks for data validity occurred by moving four conditional statements from where they currently occur in the code to where the data are initially calculated. Doing this removed redundant checks and reduced data motion. When the loops were fused we were able to contract arrays reducing the applications maximum memory footprint, the largest amount of data present in memory at any moment during execution, by 19%.

When fusing these loops we recognized we were working with a proxy application. We took care to not fuse calculations that would not be fusable in a real application due to how the code is written and their ability to handle multi-material problems. Therefore, we did not fuse the stress and hourglass calculations together. Also, in the element centered computations we kept three separate compute loops. The first handled all the computation through the *CalcMonotonicQGradientsForElems*, which includes calculations done on an element basis. Then we had a loop that in a multi-material code would go over all materials. Fused into this loop were the *MonotonicQGradients*, *EvalEOSForElems*, *MaterialApply* routines and all the loops contained in these functions. The loops to store the new volume, calculate the hydro constraint and the courant constraint were kept as separate loops. The courant and hydro calculations can be fused,

but were not because of their relatively small runtime.

There are challenges to implementing loop fusion in a multi-physics code. Fusion often results in the combination of multiple loops that are performing different calculations and can lead to maintainability and readability issues. Also, the optimal amount of fusion for one architecture or even for one problem size is not necessarily the same for another. However, it presents potentially large benefits for speeding up memory-bound codes and can reduce the memory footprint of applications that are often run at the largest size that can fit in memory. Finally, it reduces the number of times arrays must be allocated and later freed, which as the next optimization discusses can often significantly impact performance.

Global Allocation

To minimize the amount of data in memory at any given time, programmers often allocate and later destroy temporary variables. While effective at reducing the largest footprint of the code and increasing the problem size that can be run, this coding style can have significant performance impacts due to serialization on the operating system.

There are two ways the operating system can slow down codes allocating and freeing data often during execution. The serial sections of the code where the allocations and frees occur can take a significant fraction of the runtime. The runtime is due to the cost of the system call and the work of allocating memory and then freeing it. Also, lazy page allocation procedures increase translation lookaside buffer (TLB) misses. When a malloc is called, the operating system sets aside the space requested for the data, but does not setup the virtual to physical mapping of the pages. When the first element in a page is touched the operating system then sets up the mapping causing multiple TLB misses.

On Linux clusters and other systems that perform lazy allocation there are multiple ways to solve both problems. We tried the following set of techniques with LULESH. One way around both performance problems is allocating a pool of temporary work arrays at program initialization. By allocating temporaries outside of the timestep loop, only a single malloc and one setup of the translation lookaside buffer map is needed. Another solution is using large page sizes, which also reduces the number of misses and the cost of each malloc. This strategy has a similar performance impact on lazy allocated systems, but can cause memory fragmentation. An operating system feature called transparent huge pages (THP) [9] seeks to prevent fragmentation while giving the performance benefits of large pages, but is currently not used by default on Lawrence Livermore's Linux clusters due to a bug in the current implementation. Also a different malloc library was tried.

On BG/Q the only performance issue is the cost of mallocs and not lazy allocation and first touch issues. Lazy allocation is not an issue on BG/Q because a static TLB map is setup by the operating system when a job is launched for each tasks local address space, thereby alleviating the need to set one up during runtime. For BG/Q though we only tried Global Allocation.

There are a few ways the data fragmentation and/or extra data use problems can be solved other than external malloc libraries. One solution is returning to the Fortran common block programming style where users allocate a large chunk of memory and pass it between processes taking care to avoid overwriting

```
for(i = 0; i < elems; i++)
   read from corners
   function call
   non-vector code
   inner loop
   write to corners
```

```
for(i = 0; i < elems; i++)
  for(j = 0; j < vecLen; j++)
     read from corners
  for(j = 0; j < vecLen; j++)
     inlined function call
  for(j = 0; j < vecLen; j++)
     non-vector code
  for(j = 0; j < vecLen; j++)
     unrolled loop
  for(j = 0; j < vecLen; j++)
     write to corners
```

Figure 1: High level code differences between original implementation and maximally vectorized code. Original code on the left and end result on the right.

different variables. A second solution is for a program to have its own "mini-OS" that handles mallocs and frees from a common block, which was allocated at runtime. Both of these techniques would avoid freeing and allocating memory and serialization on the main OS. A final option is an operating system that handles system calls, such as mallocs and TLB misses faster, by either working in parallel or by running on more powerful cores than those performing the computation.

Increased Vectorization

The initial version of LULESH on Intel machines had 10% of its floating point instructions executed by the AVX unit on a SandyBridge machine when the icc compiler was used. Since all of the iterations of most of the loops in LULESH are fully independent, in theory, almost all the instructions can be executed by a single instruction multiple data (SIMD) unit, which is sometimes called a vector unit. To increase the vectorization of LULESH, on Linux machines we used two different non-complementary techniques.

The first way we achieved more SIMD instructions was by taking manually unrolled loops in the code and re-rolling them back into loops. By re-rolling the loops we provided more information to the icc compiler allowing it to more clearly see the dependence information. By applying this technique to six loops in *CalcElemFBHourglassForce* and one loop in *SumElemStressesToNodeForces* we were able to increase the proportion of flops executed by the vector unit from 10% to 29%. The loops that we applied this technique to were executed within an element and all had a fixed iteration bound of four or eight. Having a fixed bound makes the compiler's job easier because it can explicitly generate vector and cleanup code without worrying about profitability. While easy to apply and having a secondary benefit of code readability this technique will not be as useful in the future when longer vector widths can not be filled by the short number of loop iterations.

The second method used to increase vectorization was significantly more invasive as shown in Figure 1. Instead of working on the inner-most loops we target loops over all elements. First we defined a macro that

at compile time is set to the vector width of the target hardware for the data type being used (Single or Double). Then since most compilers only vectorize the inner most loop we unrolled all inner loops. The Intel compiler we were working with will also not vectorize an inlined function call as part of a loop so we had to inline all functions. Finally, we introduced new inner-most loops that produce SIMD instructions into the loop over the elements.

Introducing inner-most loops is necessary when parts of a loop are vectorizable and other parts are not, as shown in Figure 1. By introducing loops over the vector length defined by our macro we were able to separate vectorizable code sections from non-vectorizable ones with each loop encapsulating non-vectorizable or vectorizable code in as large a chunk as possible. Examples of vectorizable code are arithmetic instructions, min and max functions and data motion. Code that does not vectorize includes, indirect reads and writes, and some branches.

We applied our techniques to increase vectorization on the version of LULESH fused loops, because vectorization benefits compute heavy loops the most. These techniques were used on the three computationally heavy loops in each of the following functions: *IntegrateStressForElems*, *CalcFBHourglassForceForElems* and *LagrangeElements*. Just vectorizing these three loops brought the percentage of flops vectorized from 10% to 87%. We also vectorized the second loop in *LagrangeElements*, but the added compute needed to remove if statements and allow vectorization slowed the code more than the positive impact from vectorization. We did not try other loops, which are all memory bound because the benefit from vectorizing them is typically, just 2 - 10%. However, a few such as *CalcAccelerationForNodes* and the second loop in both *IntegrateStressForElems* and *CalcFBHourglassForceForElems* are vectorized by icc already.

When analyzing the applicability of these techniques to multi-physics applications the first technique is attractive to large codes because it not only increases performance in some cases, but improves code readability. However, it still requires the compiler to optimize the code to allow portable performance. The second technique is much more invasive and error prone. While a good proof of concept showing the potential of codes like LULESH to use hardware vector units, it creates a code readability nightmare by expanding the hourglass calculation from 377 lines of code spread over five functions in the rolled up loop case into 744 lines of code all within a single function. Since vectorizing C code is a significant compiler challenge it is probably easier to meet the compilers in the middle. Possible solutions include “Hybrid Index Sets” [13], a Rose tool [5] and easier to vectorize C subsets [3].

We only examined these techniques on Intel machines with icc for two reasons. The Intel compiler was better at creating SIMD instructions than others, such as gcc. Also, on BlueGene/Q the IBM compiler could not produce vectorized code from any of the techniques we used. Looking at the compiler’s output for the first vectorization technique we saw issues that included the small iteration count of the loops, non-vectorizable reductions, non-vectorizable alignment and a non-vectorizable stride issue. While all these are reasons for code not to vectorize in certain cases with our code they should not be an issue. The small loop iteration count is a plus in this case since it is statically known at compile time, the reductions are a false bottleneck due to their immediate use within the same loop so they do not need to be performed and the alignment issue can be handled by the compiler since all the variables accessed in these loops are stack allocated. Only the non-vectorizable stride is a potential problem, though with a “swizzle”

instruction these can be vectorized. However the profitability of vectorizing might not make it worthwhile.

NUMA Aware Allocation

Many modern machines have multiple memory spaces within a single node. When multiple spaces exist each processor can usually access one memory space with higher bandwidth and lower latency than the other spaces. On most machines that contain non uniform memory access (NUMA) data is allocated to the local memory of the first processor that touches it. Therefore, if problem setup occurs in serial data will not be stored locally in any processors' memory other than those that share the same memory bank with the processor performing the setup. To get around this problem the first accesses to data elements must be done in parallel using a similar access pattern to the one used during execution.

To perform NUMA aware allocation we used the Mcsup library [2], which allocates memory to the same socket that it binds tasks to. Mcsup traps all OpenMP regions and binds tasks to the same set of processors each time a parallel for loop is started. It does this with minimal programmer effort, requiring a start and end call to the library along with changing the types used for variables. We used it on Linux nodes, but not BG/Q which has uniform main memory access times.

Mcsup reduces programmer effort for pure OpenMP codes. However, for a large multi-physics code that uses MPI it is not likely needed in the near term. Large codes using a hybrid OpenMP/MPI programming model can avoid the need to properly allocate memory and pin processes by having the number of MPI tasks running on each node equal to a multiple of the number of NUMA address spaces on that node. By allocating MPI tasks in this manner the operating systems pinning of tasks will handle this work for the user.

Impact of Transformations on LULESH

T -ltr e optimizations we applied to LULESH had varying impacts on different architectures. In this section, we describe the machines and methodology that we used to test our tuned versions of LULESH. We also, explore the performance impacts of the tuning, including an analysis of how individual optimizations impact performance.

Test Methodology and Machines

Architecture	Processor Speed	Cores	Threads per Core	Flops per Cycle	Memory per Node	Memory Bandwidth
Sandybridge	2.6 GHz	16	2	4 Mult / 4 Add	32 GB	78 GB/s
BlueGene Q	1.6 GHz	16	4	8 FMA	16 GB	28.5 GB/s

Table I: Test machines used to evaluate tuning impacts.

We tested our OpenMP versions of LULESH on the machines shown in Table I. The main differences of interest are the Sandybridge machine has significantly more memory and memory bandwidth per node. Another important difference is the Sandybridge processor is an out-of-order machine, while the BG/Q

system uses in-order technology.

For our experiments we used the vendor supplied compilers, icc for Sandybridge and xlc for BlueGene Q. For icc we used version 12.1.339 with the -O3 -mavx and -openmp flags specified. For xlc we used version 12.0 and specified the -O5 -qthreaded -qsmp=omp -qsimd=noauto flags. In each case these flag combinations were chosen because they produced the best performance on the machines the compiler targeted. On BlueGene Q we set the following environment variables OMP_WAIT_POLICY=ACTIVE and BG_SMP_FAST_WAKEUP=YES. These variables enable hardware thread support and disabled threads sleeping between loops resulting in the best performance.

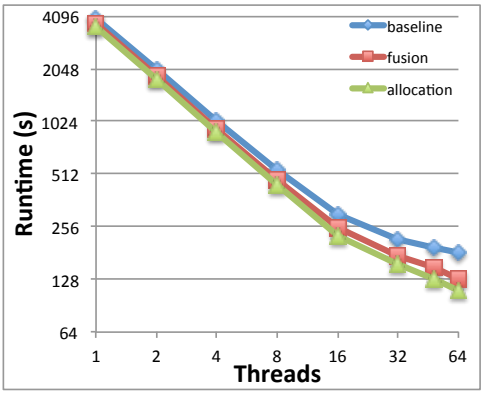
We ran three test problem sizes, 50^3, 70^3 and 90^3 on each of these architectures. These sizes were chosen to represent real problem sizes that might be run in a coupled physics simulation. While 50^3 is a bit smaller than typically used and 90^3 a bit larger we felt it was better to bound the entire space rather than pick more points within it. Each test was run five times with the minimum value selected as its runtime. Loop level data was collected in separate runs from overall runtimes.

Performance Analysis

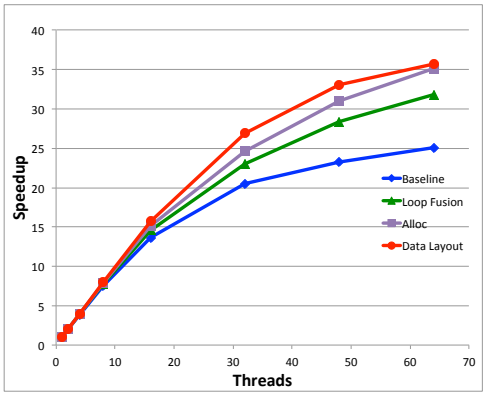
Figures 2 and 3 show the performance of the two machines on the most representative problem size 70^3. The runtime and scaling charts show the additive nature of the changes. We also show loop level data. For the baseline code the multiple loops are condensed to the single loop they are fused into in the more optimized codes. The performance of the baseline version of LULESH available online is shown in the blue line [16]. Then the red line shows the performance of that version with fused loops. Next the green line shows how global allocation and loop fusion impacts performance. On the Sandybridge machine we show the impact of adding vectorization by rolling up loops with the purple line. Finally, the cyan line shows the performance of the most aggressive vectorization. We do not show the impact of NUMA aware allocation as it only impacts the SandyBridge machine and shows up by almost halving the runtime when the second socket is used. The loop runtime data in each figure shows how each performance optimization impacts individual loops. We also include a bar for serial performance. This bar is the total program runtime minus the sum of all individual loop runtimes.

From these figures we see that loop fusion reduces the runtime on all machines and increases the strong scalability. The strong scalability is increased because there are fewer *omp parallel* regions, fewer memory-bound loops and fewer mallocs and frees. By reducing the number of *omp parallel* regions serial sections that only spawn threads are removed. The runtime on BG/Q is reduced by about 0.5 seconds on the smallest 50^3 problem and about 0.75 seconds on the larger problems. Therefore, most of the gain is due to reduced data motion and fewer mallocs not the number of *omp parallel* sections.

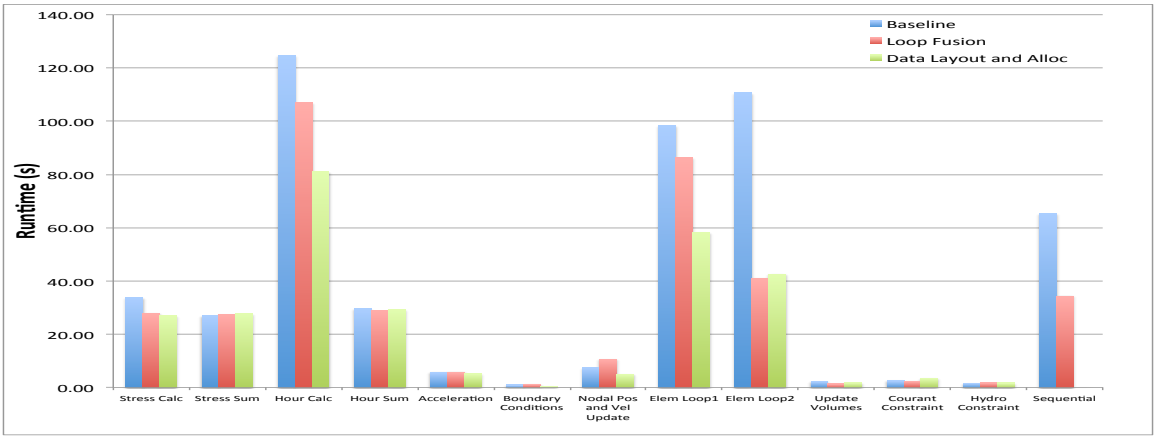
However, fusion slows down performance on both machines for the loop that updates the nodal positions and velocities. In both cases this is likely due to overwhelming the stream prefetchers. The Sandybrige processor has 32 prefetchers shared among its eight cores, while the BG/Q processor has sixteen prefetchers per core that are shared among four hardware threads. While before fusion both loops had six arrays being accessed, after fusion there are nine. Therefore, a smaller fraction of the data can be prefetched. Experiments on BG/Q showed that continuing to increase the number of streams resulted in



(a) Runtime

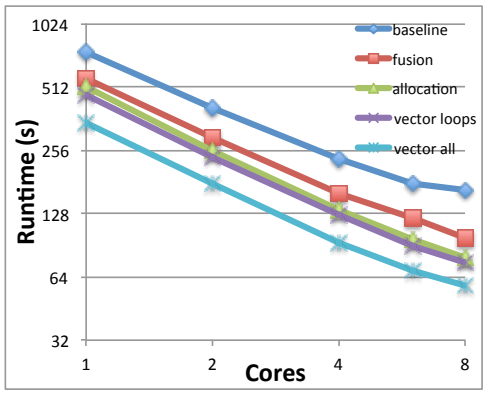


(b) Scaling

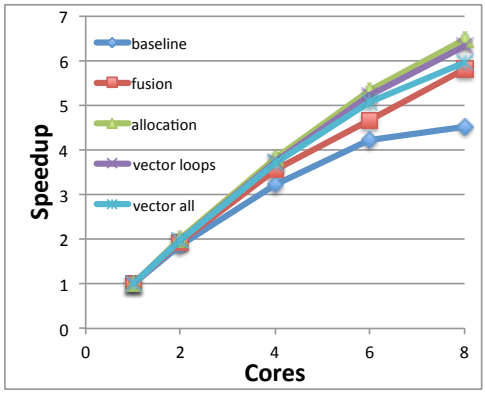


(c) Loop Level Performance

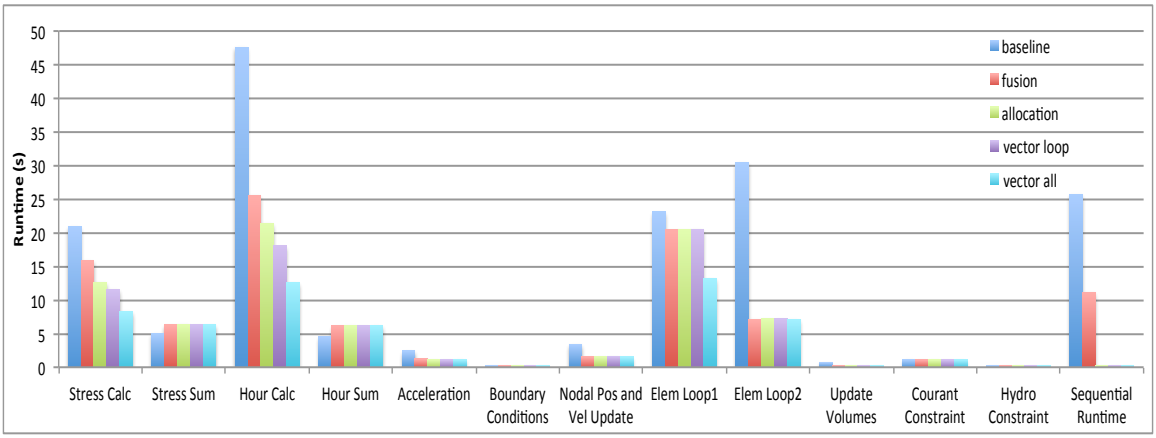
Figure 2: Performance and scaling improvement on BG/Q for a 70^3 problem.



(a) Runtime



(b) Scaling



(c) Loop Level Performance

Figure 3: Performance and scaling improvement on Sandybridge for a 70^3 problem.

worse performance as the number of streams increased beyond four with performance degradation accelerating beyond eight streams. Also, we ended up fusing a few loops that vectorize on Sandybridge with loops that do not. Since the loops that were fused in this manner were memory-bound the performance loss from no longer vectorizing was smaller than the gain from less data motion.

Fewer mallocs and frees are both an effect of loop fusion and global allocation. When both are applied the serial section of the code, not including the launching of threads, on BG/Q and other platforms becomes 0.1% or less of the total runtime. Tests run by first removing the mallocs and then fusing loops show the the effect is entirely due to getting rid of mallocs and frees. The runtime gains from removing mallocs is only significant when LULESH is run on most of the cores on a node, due to Amdahl's law, because the OS handles these misses in serial. Therefore, removing them also increases strong scalability. On the Sandybridge system removing mallocs also reduces the runtime of the loops that first touch the newly allocated data. The runtime reduction is seen in the *Stress Calc* and *Hour Calc* loops in Figure 3.

There are other ways we tried to remove the costs of the mallocs. Using large pages reduced the runtime cost of mallocs to less than a second on a 90^3 problem. The tcmalloc library from google's perf tools [4] made their cost unnoticeable. However, both of these solutions can fragment and waste memory, which can be an issue in a large multi-physics code. Transparent huge pages [9] resulted in about a seven second cost for mallocs on a 90^3 problem and will not fragment memory as much as the other solutions.

Both vectorization techniques sped up the performance on the Sandybridge machine. Rolling up loops in the hourglass and stress calculations resulted in a 6-7% overall runtime reduction. The percentage of floating point instructions vectorized also increased from 10% to 29%. The more aggressive vectorization further increased the percentage of floating point instructions vectorized resulting in 87% of all flops executed being part of a vector instruction. The more aggressive vectorization decreased runtime by 25-27% overall when compared to the optimized code without vectorization changes.

However, on BG/Q all the changes that improved vectorization on the Intel machines, called SIMDization for IBM machines, slowed down the computation. The IBM xlc compiler was unable to SIMDize any of the code that the icc compiler could. Therefore, the rolled up loops slowed down performance, by hiding other information from the compiler and the more aggressive approach probably introduced added overhead. The inability of the IBM compiler to profitably SIMDize any code is why the optimization was turned off in the xlc compiler. In fact when it was turned on the few vector instructions produced slowed down performance by up to 10%.

Overall our tuning improved performance by 2.45-2.85x on Sandybridge and 1.63-1.66x on the BG/Q system. The strong scaling of the 70^3 problem size improved from 21.69x to 32.23x on BG/Q and from 4.53x to 6.33x on Sandybridge when using all 64 hardware threads on BG/Q and eight cores on Sandybridge. Scaling improves because we reduce the percentage of time spent in memory-bound loops that improve by only 16x to 20x on BG/Q and less than 4x on Sandybridge when run on all the resources.

Changes Ported into xALE

The goal of our mini-app project was to learn lessons that can be applied back to the real code. Therefore,

we treat each successful optimization of LULESH as a potential candidate transformation to make in a real application. In our case we are using a subset of ALE3D called xALE for this experimentation and applying our changes to the Lagrange Hydro within xALE. The application of transformations to xALE has three interesting components: applying transformations to kernels represented in LULESH, applying transformations to kernels that are similar to those in LULESH, but not represented, and the relative difficulty of making transformations. In this section, we first discuss transformations we have made, followed by a discussion of the differences in making transformations.

Changes Made to xALE

One difference between xALE and LULESH is the number of hourglass routines present. LULESH has one, the FB, while xALE contains four. We have applied loop fusion to two of these hourglass routines (FB) and (GB) in both 2D and 3D. In addition, we re-rolled loops within the FB and GB to increase the vectorization. We did not pursue our more aggressive vectorization strategy because it would be difficult to maintain in a production code. We have also fused the summation of the *StressIntegrationElementLoop* and *LagrangeSumStressTerms* functions and inlined the call to *KinematicsElemLoop* within *LagrangeCalcZonal* to fuse those two loops.

We did not apply any NUMA optimizations because xALE and ALE3D are MPI only codes. MPI in these codes already constrains code to a local memory space. We also did not try global allocation since it would increase the memory footprint of the code significantly. However, we tried linking in the tcmalloc library from google's performance tools. Unlike LULESH we saw no noticeable performance impact from tcmalloc. We suspect this is because other MPI tasks hide the latency of the ones calling mallocs, setting up page tables and freeing data. Therefore, for now data allocation is unlikely to be an issue for our currently unthreaded codes. However, it could become a major performance bottleneck and we need to be cognizant of this as we thread codes for future many-core systems.

Applicability of Changes to Routines not Directly Represented

There are many functions in xALE that are significantly different from those in LULESH that could benefit from the same optimizations. While the only place we have applied changes to a kernel with different physics, is the GB hourglass filter there are other places the lessons learned in LULESH can be applied within xALE. In particular we have identified places where loop fusion can be applied. Also, there are many unrolled loops in xALE that can be re-rolled to increase vectorization.

Challenges Tuning xALE

While tuning xALE we came across significant differences from LULESH. One large obstacle to loop fusion in xALE is how multi-material problems are handled. When a loop over multiple materials is adjacent to a loop over all elements fusing them requires extra computation and possibly extra data motion to occur. For now we have only fused one set of loops in *StressIntegrationElementLoop* where extra compute is needed, but data motion is not increased. Another difference is xALE has a more complex control flow with various execution paths, while LULESH only has a single path. The multiple paths necessitated us moving conditional statements that determine the control flow to an earlier point in the

control flow to allow certain loops to be fused.

xALE Performance Improvement

We tested the performance of the changes we put into xALE on the Sandybridge machine. The BlueGene Q system was not tried as xALE currently has not been ported to it. We ran a weak scaling problem of 25^3 elements per domain. Since we are most concerned with single node performance we limited xALE to runs of 1, 8 and 64 MPI tasks. For the Sandybridge system we restricted the eight domain test case to one socket of a node. Table II shows the performance difference between the initial version of xALE and our tuned version. The runtimes reported are total xALE runtime and includes problem launch and dumping of a restart file and plot file at simulation end.

Nodes	MPI Tasks	Untuned	Tuned	Speedup
1	1	17.206	14.791	1.16
1	8	25.103	20.162	1.25
4	64	39.407	32.722	1.20

Table II: Tuned vs. Untuned xALE on Sandybridge

Overall performance improvements from the changes made to xALE result in speedups of 1.16 to 1.25x. Performance improvement was greater in parallel than in serial. These results show that as with the pure OpenMP LULESH the pure MPI xALE benefits more in parallel from the transformations we made than in serial. From these results we conclude that the lessons we learned from tuning the LULESH mini-app can be applied to a more full featured application with positive results.

Conclusions and Future Work

In this paper we described how we decrease the serial runtime of the LULESH proxy application by up to 54% and increased its scalability for modern multi-core platforms, resulting in runtime reductions of up to 65% on modern multi-core nodes. We then applied some of these same techniques to a full featured application xALE, which is a subset of ALE3D the application LULESH is meant to most closely mode. We improved the performance of xALE by up to 25% and identified other places in xALE where we can apply similar changes.

We plan to both continue working with LULESH as a place to explore new tuning ideas and coding strategies. Next, we plan to look at changing the data layouts within LULESH, by making struct of arrays to array of struct transforms. We are resurrecting the TALC compiler tool [8] that works with the ROSE compiler infrastructure to do this. Another focus of our LULESH work is determining a way to write it such that ROSE and other compiler tools can auto-tune the code. In particular we are especially interested in loop fusion and enabling code to auto-detect the optimal number of threads to use for each loop. In this work we want to maintain the current level of physics expression, so as to not create development and maintainability issues. If successful this will allow cross platform performance portability. However, it will not be without challenges as tuning parameters in real codes might vary based on not just the architecture run on, but also the problem being run. Solving this problem might require compiling the code

on a per problem or per problem class basis, which will require research into the costs and benefits of this.

We plan to continue porting successful code changes to LULESH back to xALE further showing their applicability. We also will look at the optimizations relevance to GPUs. Our proxy-app work will continue with the advection (LUAU3D) and radiation diffusion (MULARD) mini-apps where we will see if similar performance optimizations are useful. In addition, we plan to explore applying the lessons learned tuning LULESH to other full featured codes. Applying what we learned with one proxy-app to a completely different code will allow us to access the ability of a single proxy to represent many codes.

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.0 (LLNL-CONF-610032)

References

[1] Hydrodynamics Chllange Problem. Livermore National Labarotory, Livermore, CA, LLNL-TR-490254.

[2] A.H. Baker, T. Gamblin, M. Schulz, and U.M. Yang. Challenges of Scaling Algebraic Multigrid Across Modern Multicore Architectures. In Proceedings of the 25th IEEE International Parallal and Distrbuted Processing Symposium (IPDPS 2011), Anchorage, AK, May 16-20, 2011. LLNL-CONF-458074.

[3] Ronald Lei, Sebastian Hack and Ingo Wald. Extending a C-like language for portable SIMD programming. Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, New Orleans, Louisiana, USA, February, 2012 pages 65-74.

[4] Sanjay Ghemawat. ATCMalloc: Thread-Caching Malloc. 2012, http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[5] D. Quinlan. ROSE: Compiler support for object-oriented frameworks. Parallel Processing Letters, World Scientific, 10:02n03, 2000, pages 215–226.

[6] G. Gao, R. Olson, V. Sarkar and R. Thekkath. Collective loop fusion for array contraction. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August, 2004, pages 281-295.

[7] Elizabeth R. Jessup, Ian Karlin, Erik Silkensen, Geoffrey Belter and Jeremy Siek. Understanding memory effects in the automated generation of optimized matrix algebra kernels. Procedia Computer Science, Volume 1, Issue 1, May 2010, Pages 1873-1881.

[8] J Keasler, T Jones, and D Quinlan. TALC: A Simple C Language Extension For Improved Performance and Code Maintainability. In *Proccesdings of the 9th LCI International Conference on High-Performance Clustered Computing*, Urbana, Il, April 28-May 1, 2008.

[9] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. SIGOPS Oper. Syst. Rev. 36, SI (December 2002), pages 89-104.

[10] H. U. Mair. Review: Hydrocodes for structural response to underwater explosions. Shock and Vibration, 6:2, 1999, IOS Press, pages 81–96.

[11] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz and Charles Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In Procceedings of 27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013), Boston, USA, May, 2013 (to appear).

[12] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk and Jesper Larsson Traff. MPI at Exascale. In Processings of Scientific Discovery through Advanced Computing (SciDAC 2010), Chattanooga, Tennessee, July, 2010.

[13] Rich Hornung and Jeff Keasler. Personal Communication. December 2012.

[14] Andrew I. Stone, John M. Dennis and Michelle Mills Strout.. Evaluating Coarray Fortran with the CGPOP Miniapp. Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS), October, 2011.

[15] Ale3D. 2012, https://wci.llnl.gov/codes/ale3d/.

[16] Codesign at Lawrence Livermore National Laboratory. 2013, https://codesign-dev.llnl.gov/lulesh.php.